Práctica 3. Programación con tareas y eventos en ESP-IDF

Objetivos

El objetivo de esta práctica es conocer los mecanismos para la gestión de tareas que ofrece FreeRTOS, concretamente en su porting ESP-IDF (con alguna particularidad por el hecho de estar adaptado a tener 2 cores).

Trabajaremos los siguientes aspectos del API de ESP-IDF:

- Familiarizarse con la API de *tareas* y *eventos* en ESP/IDF.
- Uso de *delays* para tareas periódicas (veremos mejores opciones en el futuro)
- Comunicación y sincronización de tareas mediante colas

Material de consulta

Para ver los detalles de cada aspecto de esta práctica se recomienda la lectura de los siguientes enlaces:

- API de ESP-IDF
- Documentación oficial de FreeRTOS
- Mastering de FreeRTOS Real Time Kernel

Introducción

Al desarrollar código para sistemas empotrados, como nuestro nodo basado en ESP32, es habitual organizar la aplicación en torno a diferentes tareas que se ejecutan de forma concurrente. Habrá tareas dedicadas al muestreo de sensores, tareas dedicadas a la conectividad, tareas de *logging*...

Por tanto, al comenzar un desarrollo con un nuevo *RTOS* (Real-Time Operating System) es importante conocer qué servicios ofrece el sistema para la gestión de hilos/tareas. En ocasiones, puede no haber ningún soporte. En otras ocasiones, el API ofrecida será específica del sistema operativo utilizado (como es el caso con FreeRTOS y, por tanto, con la extensión que usaremos: ESP-IDF). Y, en ocasiones, el sistema ofrecerá algún API estándar, como el de POSIX.

En los vídeos y transparencias de la asignatura disponibles en el Campus Virtual se hace una breve introducción de los mecanismos de:

- Creación y destrucción de tareas en ESP-IDF.
- Comunicación y sincronización de tareas mediante colas
- Uso de *eventos* como sistema de comunicación asíncrona.

Los siguientes ejercicios se proponen como una práctica sencilla de esos mecanismos.

Ejercicios básicos

Muestreo periódico del sensor de efecto Hall

Un sensor de efecto Hall permite la medición de campos magnéticos o corrientes. En este ejercicio no entraremos a ver los detalles del sensor en sí y simplemente estamos interesados en su uso en el entorno ESP-IDF. El Hall sensor está conectado a un canal del *ADC* (Conversor Analógico-Digital) que estudiaremos más adelante. Por ahora, nos basta con saber que para realizar una lectura del sensor basta con invocar a la función `hall_sensor_read()` y que, previamente, es necesario configurar el ADC mediante la llamada `adc1_config_width(ADC_WIDTH_12Bit)` (sólo es necesario invocar esta llamada una vez).

```
#include <driver/adc.h>
...

    adc1_config_width(ADC_WIDTH_12Bit);
    int val = hall_sensor_read();
```

En este primer ejercicio NO crearemos más tareas y simplemente usaremos un bucle infinito en la función `app_main()` para leer de forma periódica el sensor. Asimismo, usaremos la llamada `void vTaskDelay(const TickType_t xTicksToDelay)` para realizar las esperas entre lecturas.

Tarea

Crea una aplicación que lea el valor del sensor de efecto Hall cada 2 segundos y muestre el valor leído por puerto serie.

Cuestión

¿Qué prioridad tiene la tarea inicial que ejecuta la función `app_main()`? ¿Con qué llamada de ESP-IDF podemos conocer la prioridad de una tarea?

Creación de una tarea para realizar el muestreo

Modifica el código anterior para crear una nueva tarea que sea la encargada de realizar el muestreo (denominaremos *muestreadora* a dicha tarea). La tarea muestreadora comunicará la lectura con la tarea inicial (la que ejecuta `app_main()`) a través de una variable global.

Tarea

La tarea creada leerá el valor del sensor de efecto Hall con un período que se pasará como argumento a la tarea. La tarea inicial recogerá ese valor y lo mostrará por puerto serie.

Cuestión

- ¿Cómo sincronizas ambas tareas?¿Cómo sabe la tarea inicial que hay un nuevo dato generado por la tarea muestreadora?
- Si además de pasar el período como parámetro, quisiéramos pasar como argumento la dirección en la que la tarea muestreadora debe escribir las lecturas, ¿cómo pasaríamos los dos argumentos a la nueva tarea?

Comunicación mediante colas

Modifica el código anterior para que las dos tareas (inicial y muestreadora) se comuniquen mediante una cola de ESP-IDF.

Tarea

La tarea creada (muestreadora) recibirá como argumento el período de muestreo y la cola en la que deberá escribir los datos leídos.

Cuestión

Al enviar un dato por una cola, ¿el dato se pasa por copia o por referencia?. Consulta la documentación para responder.

Uso de eventos

Finalmente, se modificará nuevamente el código de muestreo original (no el que usa una cola para comunicar) para que utilice eventos para notificar que hay una nueva lectura que mostrar por el puerto serie.

Para ello se declara un nuevo *event base* llamado *HALL_EVENT* y al menos un `event ID` que se denominará `HALL_EVENT_NEWSAMPLE`.

Tarea

La tarea creada (muestreadora) recibirá como argumento el período de muestreo. Cuando tenga una nueva muestra, la comunicará a través de `esp_event_post_to()`. La tarea inicial registrará un `handler` que se encargará de escribir en el puerto serie.

Cuestión

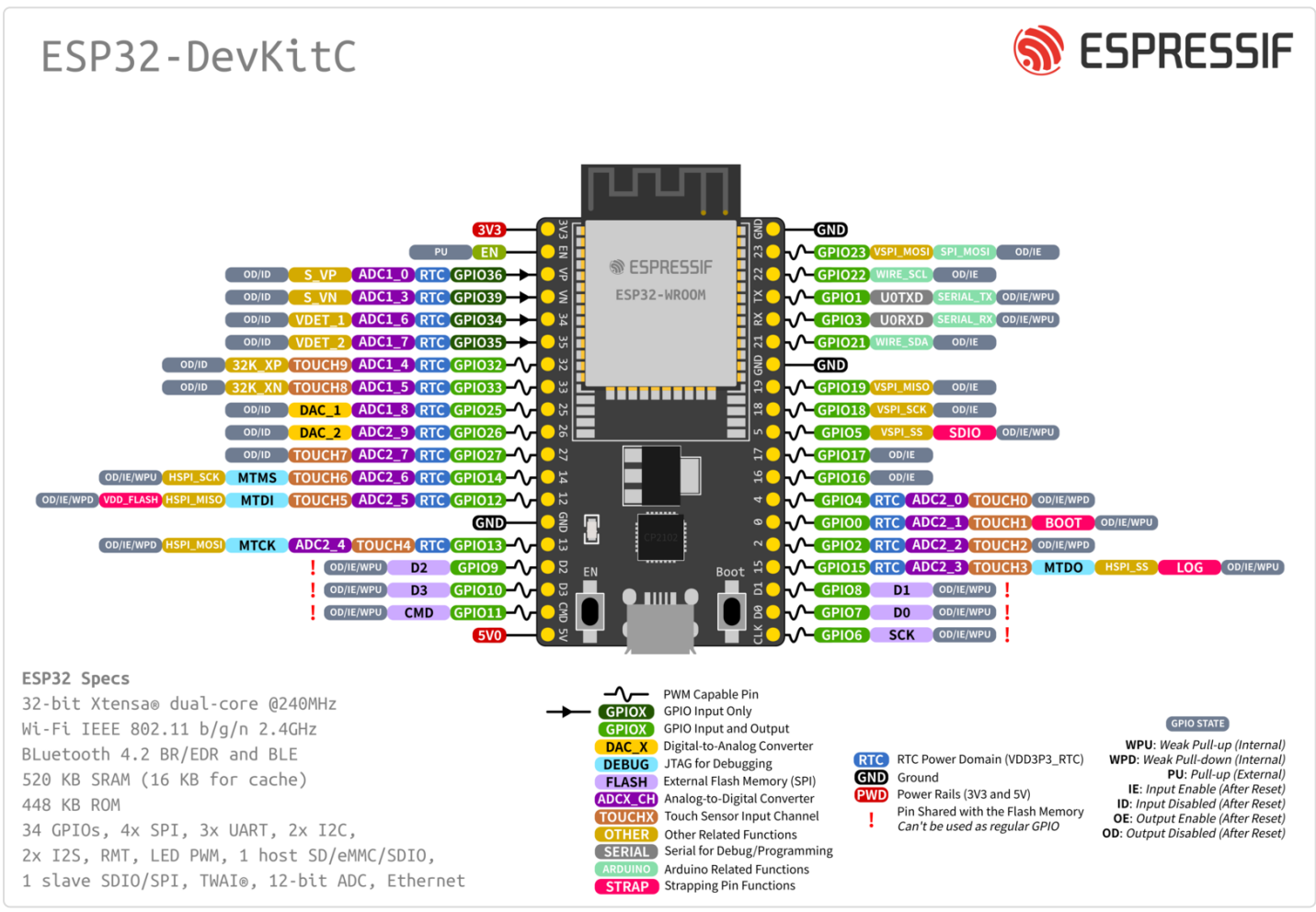
¿Qué debe hacer la tarea inicial tras registrar el *handle*? ¿Puede finalizar?

Controladores GPIO

Los controladores de GPIO (*General Purpose Input-Ouput*) permiten controlar ciertos pines de nuestro dispositivo para usarlos como entrada (por ejemplo, para conectar un botón) o salida (por ejemplo para conectar un LED) o con funciones *especiales* (que forme parte de un bus serie, por ejemplo).

El SoC ESP32 que usamos proporciona 40 GPIO *pads* (el SoC no tiene pines propiamente dichos, sino conectores, normalmente de superfície, que se deminan *pad*). El módulo WROOM-32 que usamos expone 38 de ellos, que son accesibles a traves de los pines (los conectores físicos a ambos lados de la placa) que incorpora nuestra placa DevKitC.

En la siguiente figura se muestra la disposición de los pines en la placa ESP32-DevKitC que usamos en nuestras prácticas:



En [la web de Espressif] se pueden encontrar más detalles de la placa.

[Como se indica en la documentación de ESP-IDF], algunos de esos pines tiene un propósito específico. Por ejemplo, GPIO6-11 y 16-17 no deben usarse porque están internamente conectados a la memoria SPI flash. También nos indica que los pines del canal 2 del ADC (ADC2) NO deben usarse mientras se utiliza Wi-Fi. Es muy conveniente leer todas las restricciones para evitar problemas en nuestros desarrollos.

La documentación muestra también la API que ofrece ESP-IDF para configurar los pines (entrada o salida, uso de pull-up/pull-down) establecer un valor lógico (0 ó 1)en un pin (previamente configurado como salida) o leer el valor lógico de un pin (configurado como entrada).

El siguiente código, [extraído del ejemplo de GPIO proporcionado en la distribución ESP-IDF], muestra cómo configurar los pines GPIO18 y GPIO19 como salida. Observa cómo se construye la máscara de bits `GPIO_OUTPUT_PIN_SEL` para indicar a `gpio_config()` qué pines se configuran.

```
#define GPIO_OUTPUT_IO_0 18
#define GPIO_OUTPUT_IO_1 19
#define GPIO_OUTPUT_PIN_SEL ((1ULL<<GPIO_OUTPUT_IO_0) | (1ULL<<GPIO_OUTPUT_IO_1))
gpio_config_t io_conf;
io_conf.intr_type = GPIO_PIN_INTR_DISABLE;
io_conf.mode = GPIO_MODE_OUTPUT;
io_conf.pin_bit_mask = GPIO_OUTPUT_PIN_SEL;
io_conf.pull_down_en = 0;
io_conf.pull_up_en = 0;
gpio_config(&io_conf);
```

Posteriormente, podemos establecer el valor lógico de la salida con una llamada similar a `gpio_set_level(GPIO_OUTPUT_IO_1, valor);`, siendo `valor` igual a 0 ó 1.

De forma similar el siguiente código configura los pines 4 y 5 como entrada:

```
    #define GPIO_INPUT_IO_0 4
    #define GPIO_INPUT_IO_1 5
    #define GPIO_INPUT_PIN_SEL ((1ULL<<GPIO_INPUT_IO_0) | (1ULL<<GPIO_INPUT_IO_1))
    gpio_config_t io_conf;
    io_conf.pin_bit_mask = GPIO_INPUT_PIN_SEL;
    io_conf.mode = GPIO_MODE_INPUT;
    gpio_config(&io_conf);
```

Posteriormente, podremos leer el valor lógico de esos pines con una llamada a `gpio_get_level()`.

Segunda sesión: ejercicio de estructuración de código

Queremos montar un sistema monitorice la temperatura y humedad con un cierto período, y envíe los datos por red. Así mismo, monitorizará la pulsación de un botón para detectar la cercanía de un operador, lo que llevará al sistema a un modo de funcionamiento diferente, mostrando una consola por puerto serie.

La funcionalidad del sistema será la siguiente:

- El sistema tendrá dos modos de funcionamiento: `monitorización` o `consola`. En el primero, se monitorizará y enviará la temperatura y humedad. En el segundo se permitirá el uso de una consola de comandos.
- Monitorizará la temperatura y la humedad cada `n` segundos, siendo éste un parámetro seleccionable por `menuconfig`. La lectura de cada medida se *comunicará mediante eventos*. Todo el código relacionado con las lecturas del sensor estará en un componente separado. Se valorará la modularización del código (uso de más componentes) para el resto de funcionalidad de este punto.
- Las lecturas se enviarán por red mediante WiFi. En este caso, no usaremos realmente la WiFi pero se programará un componente que lo simulre. Ofrecerá un API similar a:
    - `wifi_connect()` trata de conectar a WiFi. Cuando la conexión se produce, recibiremos un evento. Una vez conseguida, tratará de conseguir una IP (sin que hagamos ninguna otra llamada) y recibiremos un evento al conseguirla.
    - `wifi_disconnect()`. Desconecta de la WiFi.
    - `esp_err_t send_data_wifi(void* data, size_t size)`. Permite enviar un dato mediante la conexión WiFI. Devolverá un error si el envío no se pudo realizar. Imprimirá el dato por puerto serie (pasaremos siempre una cadena de caracteres como dato de entrada).

  El componente enviará los siguientes eventos: * `WIFI_CONECTADO`. Se enviará cuando el módulo de WiFi consiga conexión (equivalente a conectar al SSID) * `WIFI_DESCONECTADO`. Se enviará cuando se pierda la conectividad. Será necesario llamar a `wifi_connect()` nuevamente para volver a conectar. * `IP_CONSEGUIDA`. Se enviará cuando se haya conseguido IP. Hasta entonces, la aplicación no debería llara a `send_data_wifi` pues fallará siempre. En otro caso, esa llamada siempre dará éxito.

  El componente simulará la conexión WiFI y tendrá un parámetro que modelará la latencia de conexión (tiempo desde que se llama a `wifi_connect()` hasta que se conecta a WiFi), latencia para conseguir IP y la tasa de desconexión (probabilidad de que la conexión falle y

tengamos que volver a conectar. Se modelará como un número de segundos tras la conexión)
- Cuando no se disponga de conectividad WiFi, las lecturas del sensor se seguirán realizando, pero se almacenarán en memoria Flash. Para simular esta parte, se creará un nuevo componente que emulará el uso de la memoria Flash mediante el siguiente API:
  - `esp_err_t writeToFlash(void* data, size_t size)` permite escribir en memoria flash el dato `data` de tamaño `size` bytes La siguiente llamada a `writeToFlash()` escribirá a continuación del dato anterior sin sobreescribirlo.
  - `void* readFromFlash(size_t size)`. Lee el dato (`size` bytes) más antiguo almacenado en la flash. Esos bytes quedan marcados como leídos y se podrán usar en futuras escrituras.
  - `size_t getDataLeft()` nos devuelve cuántos bytes hay pendientes de ser leídos en la flash.

  Se realizará un componente que emule así el comportamiento de una memoria Flash. Como simplificación, se asumirá que el tamaño de lectura/escritura siempre será el mismo (el tamaño de un `float`, que será el tipo usado en las lecturas del sensor). Los datos se almancenarán en un buffer circular (no hay que usar la flash de verdad).

- Cuando consigamos conexión (WiFi + IP), enviaremos los datos que tengamos pendientes en la memoria flash (si hay alguno).
- La aplicación monitorizará (cada `nhall` segundos; parametrizable) un pin de GPIO para detectar pulsaciones de un botón. Si se produce una pulsación, pasaremos al modo `consola`. Se escribirá un componente para esta funcionalida. Si se detecta una pulsación, *se enviará un evento*.

- Cuando estemos en el modo `consola` se utilizará el componente `consola` de ESP-IDF para leer comandos del usuario. En concreto habrá 3 comandos disponibles:
  - `help` que mostrará los comandos disponibles
  - `monitor` que volverá nuevamente al modo `monitorización`, tratando de conectar a WiFi de nuevo.
  - `quota` que nos informará de cuántos bytes tiene ocupadas la flash simulada (es decir, cuántos no se han leído)

  Durante este modo de funcionamiento, no se monitorizará el sensor Si7021 y nos desconetaremos de la WiFi.

Tarea

Escribe una aplicación que realice la funcionalidad anterior. Se valorará especialmente la modularidad y estructura del código, de modo que sea extensible y reutilizable.
